Two views of syntactic structure
Two views of linguistic structure:
1. Constituency (phrase structure)

- Phrase structure organizes words into nested constituents.
- How do we know what is a constituent? (Not that linguists don’t argue about some cases.)
  - Distribution: a constituent behaves as a unit that can appear in different places:
    - John talked [to the children] [about drugs].
    - John talked [about drugs] [to the children].
    - *John talked drugs to the children about
  - Substitution/expansion/pro-forms:
    - I sat [on the box/right on top of the box/there].
  - Coordination, regular internal structure, semantics, ...

[Diagram showing phrase structure analysis]
Analysts said -NONE- to Mr. Stronach wants NP-SBJ to resume *-1 to more influential role in NP-SBJ running DT the NN company
Two views of linguistic structure:
2. Dependency structure

- Dependency structure shows which words depend on (modify or are arguments of) which other words.
Statistical Natural Language Parsing

Parsing: The rise of data and statistics
Pre 1990 (“Classical”) NLP Parsing

- Wrote symbolic grammar (CFG or often richer) and lexicon
  
  \[
  \begin{align*}
  S &\rightarrow NP \ VP \\
  NP &\rightarrow (DT) \ NN \\
  NP &\rightarrow NN \ NNS \\
  NP &\rightarrow NNP \\
  VP &\rightarrow V \ NP \\
  NN &\rightarrow interest \\
  NNS &\rightarrow rates \\
  NNS &\rightarrow raises \\
  VBP &\rightarrow interest \\
  VBZ &\rightarrow rates
  \end{align*}
  \]

- Used grammar/proof systems to prove parses from words

- This scaled very badly and didn’t give coverage. For sentence:

  *Fed raises interest rates 0.5% in effort to control inflation*

  - Minimal grammar: 36 parses
  - Simple 10 rule grammar: 592 parses
  - Real-size broad-coverage grammar: millions of parses
Classical NLP Parsing: The problem and its solution

• Categorical constraints can be added to grammars to limit unlikely/weird parses for sentences
  • But the attempt make the grammars not robust
    • In traditional systems, commonly 30% of sentences in even an edited text would have no parse.
• A less constrained grammar can parse more sentences
  • But simple sentences end up with ever more parses with no way to choose between them
• We need mechanisms that allow us to find the most likely parse(s) for a sentence
  • Statistical parsing lets us work with very loose grammars that admit millions of parses for sentences but still quickly find the best parse(s)
The rise of annotated data: The Penn Treebank

[Marcus et al. 1993, Computational Linguistics]
The rise of annotated data

Starting off, building a treebank seems a lot slower and less useful than building a grammar.

But a treebank gives us many things:
- Reusability of the labor
  - Many parsers, POS taggers, etc.
  - Valuable resource for linguistics
- Broad coverage
- Frequencies and distributional information
- A way to evaluate systems
Statistical parsing applications

Statistical parsers are now robust and widely used in larger NLP applications:

• High precision question answering [Pasca and Harabagiu SIGIR 2001]
• Improving biological named entity finding [Finkel et al. JNLPBA 2004]
• Syntactically based sentence compression [Lin and Wilbur 2007]
• Extracting opinions about products [Bloom et al. NAACL 2007]
• Improved interaction in computer games [Gorniak and Roy 2005]
• Helping linguists find data [Resnik et al. BLS 2005]
• Source sentence analysis for machine translation [Xu et al. 2009]
• Relation extraction systems [Fundel et al. Bioinformatics 2006]
Attachment ambiguities

- A key parsing decision is how we ‘attach’ various constituents
  - PPs, adverbial or participial phrases, infinitives, coordinations, etc.

The board approved [its acquisition] [by Royal Trustco Ltd.]
  [of Toronto]
  [for $27 a share]
  [at its monthly meeting].

- Catalan numbers: $C_n = \frac{(2n)!}{[(n+1)!n!]}$
- An exponentially growing series, which arises in many tree-like contexts:
  - E.g., the number of possible triangulations of a polygon with $n+2$ sides
    - Turns up in triangulation of probabilistic graphical models….
Two problems to solve:
1. Repeated work...
Two problems to solve:
2. Choosing the correct parse

- How do we work out the correct attachment:
  - She saw the man with a telescope
  - Is the problem ‘AI complete’? Yes, but ...
  - Words are good predictors of attachment
    - Even absent full understanding
  - Moscow sent more than 100,000 soldiers into Afghanistan ...
  - Sydney Water breached an agreement with NSW Health ...

- Our statistical parsers will try to exploit such statistics.
CFGs and PCFGs

(Probabilistic) Context-Free Grammars
A phrase structure grammar

S → NP VP
VP → V NP
VP → V NP PP
NP → NP NP
NP → NP PP
NP → N
NP → e
PP → P NP

N → people
N → fish
N → tanks
N → rods
V → people
V → fish
V → tanks
P → with

people fish tanks
people fish with rods
Phrase structure grammars = context-free grammars (CFGs)

- $G = (T, N, S, R)$
  - $T$ is a set of terminal symbols
  - $N$ is a set of nonterminal symbols
  - $S$ is the start symbol ($S \in N$)
  - $R$ is a set of rules/productions of the form $X \rightarrow \gamma$
    - $X \in N$ and $\gamma \in (N \cup T)^*$

- A grammar $G$ generates a language $L$. 
Phrase structure grammars in NLP

- $G = (T, C, N, S, L, R)$
  - $T$ is a set of terminal symbols
  - $C$ is a set of preterminal symbols
  - $N$ is a set of nonterminal symbols
  - $S$ is the start symbol ($S \in N$)
  - $L$ is the lexicon, a set of items of the form $X \rightarrow x$
    - $X \in C$ and $x \in T$
  - $R$ is the grammar, a set of items of the form $X \rightarrow \gamma$
    - $X \in N$ and $\gamma \in (N \cup C)^*$

- By usual convention, $S$ is the start symbol, but in statistical NLP, we usually have an extra node at the top (ROOT, TOP)
- We usually write $e$ for an empty sequence, rather than nothing
Probabilistic – or stochastic – context-free grammars (PCFGs)

- $G = (T, N, S, R, P)$
  - $T$ is a set of terminal symbols
  - $N$ is a set of nonterminal symbols
  - $S$ is the start symbol ($S \in N$)
  - $R$ is a set of rules/productions of the form $X \rightarrow \gamma$
  - $P$ is a probability function
    - $P: R \rightarrow [0,1]$
    - $\forall X \in N, \sum_{X \rightarrow \gamma \in R} P(X \rightarrow \gamma) = 1$

- A grammar $G$ generates a language model $L$.

$$\sum_{\gamma \in T^*} P(\gamma) = 1$$
A PCFG

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td>1.0</td>
</tr>
<tr>
<td>VP → V NP</td>
<td>0.6</td>
</tr>
<tr>
<td>VP → V NP PP</td>
<td>0.4</td>
</tr>
<tr>
<td>NP → NP NP</td>
<td>0.1</td>
</tr>
<tr>
<td>NP → NP PP</td>
<td>0.2</td>
</tr>
<tr>
<td>NP → N</td>
<td>0.7</td>
</tr>
<tr>
<td>PP → P NP</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Terminal</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>people</td>
<td>0.5</td>
</tr>
<tr>
<td>N</td>
<td>fish</td>
<td>0.2</td>
</tr>
<tr>
<td>N</td>
<td>tanks</td>
<td>0.2</td>
</tr>
<tr>
<td>N</td>
<td>rods</td>
<td>0.1</td>
</tr>
<tr>
<td>V</td>
<td>people</td>
<td>0.1</td>
</tr>
<tr>
<td>V</td>
<td>fish</td>
<td>0.6</td>
</tr>
<tr>
<td>V</td>
<td>tanks</td>
<td>0.3</td>
</tr>
<tr>
<td>P</td>
<td>with</td>
<td>1.0</td>
</tr>
</tbody>
</table>

[With empty NP removed so less ambiguous]
The probability of trees and strings

- $P(t)$ – The probability of a tree $t$ is the product of the probabilities of the rules used to generate it.
- $P(s)$ – The probability of the string $s$ is the sum of the probabilities of the trees which have that string as their yield

$$P(s) = \sum_j P(s, t) \text{ where } t \text{ is a parse of } s$$

$$= \sum_j P(t)$$
$t_1:$

```
S_{1.0}
  └── VP_{0.4}
      └── PP_{1.0}
          └── NP_{0.7}

  └── V_{0.6}
      └── NP_{0.7}
          └── N_{0.2}
              └── tanks

  └── NP_{0.7}
      └── N_{0.1}
          └── rods
      └── with
```

```
NP_{0.7}

N_{0.5}
people

fish
```

```
S_{1.0}
```
$t_2$: $S_{1.0}$

```
    NP_{0.7}
   /   |
N_{0.5} V_{0.6} NP_{0.2}
     /   |
  people fish NP_{0.7} PP_{1.0}
       /   |
  tanks with NP_{0.7} N_{0.1}
             |
     rods
```
Tree and String Probabilities

- \( s = \text{people fish tanks with rods} \)
- \( P(t_1) = 1.0 \times 0.7 \times 0.4 \times 0.5 \times 0.6 \times 0.7 \times 1.0 \times 0.2 \times 1.0 \times 0.7 \times 0.1 \)
  \[= 0.0008232\]
- \( P(t_2) = 1.0 \times 0.7 \times 0.6 \times 0.5 \times 0.6 \times 0.2 \times 0.7 \times 1.0 \times 0.2 \times 1.0 \times 0.7 \times 0.1 \)
  \[= 0.00024696 \quad \text{[more depth }\rightarrow\text{ small number]}\]
- \( P(s) = P(t_1) + P(t_2) \)
  \[= 0.0008232 + 0.00024696 \]
  \[= 0.00107016\]
Chomsky Normal Form

- All rules are of the form $X \rightarrow YZ$ or $X \rightarrow w$
  - $X, Y, Z \in N$ and $w \in T$
- A transformation to this form doesn’t change the weak generative capacity of a CFG
  - That is, it recognizes the same language
    - But maybe with different trees (strong)
- Empties and unaries are removed recursively
- $n$-ary rules are divided by introducing new nonterminals ($n > 2$)
A phrase structure grammar

S → NP VP
VP → V NP
VP → V NP PP
NP → NP NP
NP → NP PP
NP → N
NP → e
PP → P NP

N → people
N → fish
N → tanks
N → rods
V → people
V → fish
V → tanks
P → with
Chomsky Normal Form

S → NP VP
VP → V NP
S → V NP
VP → V @VP_V
@VP_V → NP PP
S → V @S_V
@S_V → NP PP
VP → V PP
S → V PP
NP → NP NP
NP → NP PP
NP → P NP
PP → P NP

NP → people
NP → fish
NP → tanks
NP → rods
V → people
V → fish
S → people
VP → people
V → fish
S → fish
VP → fish
V → tanks
S → tanks
VP → tanks
P → with
PP → with
Chomsky Normal Form

- You should think of this as a transformation for efficient parsing
- With some extra book-keeping in symbol names, you can even reconstruct the same trees with a detransform
- In practice full Chomsky Normal Form is a pain
  - Reconstructing n-aries is easy
  - Reconstructing unaries/empties is trickier

- **Binarization** is crucial for cubic time CFG parsing

- The rest isn’t necessary; it just makes the algorithms cleaner and a bit quicker
CKY Parsing

Exact polynomial time parsing of (P)CFGs
Constituency Parsing

PCFG

Rule Prob $\theta_i$

$S \rightarrow NP \ VP \quad \theta_0$

$NP \rightarrow NP \ NP \quad \theta_1$

$\ldots$

$N \rightarrow fish \quad \theta_{42}$

$N \rightarrow people \quad \theta_{43}$

$V \rightarrow fish \quad \theta_{44}$

$\ldots$
Cocke-Kasami-Younger (CKY)
Constituency Parsing

0 fish 1 people 2 fish 3 tanks 4
Viterbi (Max) Scores

\[
\begin{align*}
S &\rightarrow NP\ VP & 0.9 \\
S &\rightarrow VP & 0.1 \\
VP &\rightarrow V\ NP & 0.5 \\
VP &\rightarrow V & 0.1 \\
VP &\rightarrow V\ @VP_V & 0.3 \\
@VP_V &\rightarrow NP\ PP & 1.0 \\
NP &\rightarrow NP\ NP & 0.1 \\
NP &\rightarrow NP\ PP & 0.2 \\
NP &\rightarrow N & 0.7 \\
PP &\rightarrow P\ NP & 1.0
\end{align*}
\]
Viterbi (Max) Scores

\[ S = \max S' \text{ (all } S' \text{ in } S) \]
\[ VP = \max VP' \text{ (all } VP' \text{ in } VP) \]

<table>
<thead>
<tr>
<th></th>
<th>people</th>
<th>fish</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>VP</td>
<td></td>
<td>0.06</td>
</tr>
<tr>
<td>NP</td>
<td>0.14</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>0.2</td>
<td></td>
</tr>
</tbody>
</table>
Extended CKY parsing

• Unaries can be incorporated into the algorithm
  • Messy, but doesn’t increase algorithmic complexity

• Empties can be incorporated
  • Use fenceposts – [0 people 1 fish 2 tank 3 fish 4]
  • Doesn’t increase complexity; essentially like unaries

• Binarization is *vital*
  • Without binarization, you don’t get parsing cubic in the length of the sentence and in the number of nonterminals in the grammar
    • Binarization may be an explicit transformation or implicit in how the parser works *(Earley-style dotted rules)*, but it’s always there.
The CKY algorithm (1960/1965)  
... extended to unaries

function CKY(words, grammar) returns [most_probable_parse,prob]
  score = new double[#(words)+1][#(words)+1][#(nonterms)]
  back = new Pair[#(words)+1][#(words)+1][#nonterms]
  for i=0; i<#(words); i++
    for A in nonterms
      if A -> words[i] in grammar
        score[i][i+1][A] = P(A -> words[i])
  //handle unaries
  boolean added = true
  while added
    added = false
    for A, B in nonterms
      if score[i][i+1][B] > 0 && A->B in grammar
        prob = P(A->B)*score[i][i+1][B]
        if prob > score[i][i+1][A]
          score[i][i+1][A] = prob
          back[i][i+1][A] = B
          added = true
The CKY algorithm (1960/1965) ... extended to unaries

for span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
      for A,B,C in nonterms
        prob=score[begin][split][B]*score[split][end][C]*P(A→BC)
        if prob > score[begin][end][A]
          score[begin][end][A] = prob
          back[begin][end][A] = new Triple(split,B,C)
    //handle unaries
    boolean added = true
    while added
      added = false
      for A, B in nonterms
        prob = P(A→B)*score[begin][end][B];
        if prob > score[begin][end][A]
          score[begin][end][A] = prob
          back[begin][end][A] = B
          added = true
    return buildTree(score, back)
The grammar:
Binary, no epsilons,

\[
\begin{align*}
S & \rightarrow NP \ VP \quad 0.9 \\
S & \rightarrow VP \quad 0.1 \\
VP & \rightarrow V \ NP \quad 0.5 \\
VP & \rightarrow V \quad 0.1 \\
VP & \rightarrow V \ @VP_V \quad 0.3 \\
VP & \rightarrow V \ PP \quad 0.1 \\
@VP_V & \rightarrow NP \ PP \quad 1.0 \\
NP & \rightarrow NP \ NP \quad 0.1 \\
NP & \rightarrow NP \ PP \quad 0.2 \\
NP & \rightarrow N \quad 0.7 \\
PP & \rightarrow P \ NP \quad 1.0 \\
N & \rightarrow people \quad 0.5 \\
N & \rightarrow fish \quad 0.2 \\
N & \rightarrow tanks \quad 0.2 \\
N & \rightarrow rods \quad 0.1 \\
V & \rightarrow people \quad 0.1 \\
V & \rightarrow fish \quad 0.6 \\
V & \rightarrow tanks \quad 0.3 \\
P & \rightarrow with \quad 1.0
\end{align*}
\]
<table>
<thead>
<tr>
<th></th>
<th>fish</th>
<th>1</th>
<th>people</th>
<th>2</th>
<th>fish</th>
<th>3</th>
<th>tanks</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>score[0][1]</td>
<td>score[0][2]</td>
<td>score[0][3]</td>
<td>score[0][4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
S → NP VP 0.9
S → VP 0.1
VP → V NP 0.5
VP → V 0.1
VP → V @VP_V 0.3
VP → V PP 0.1
@VP_V → NP PP 1.0
NP → NP NP 0.1
NP → NP PP 0.2
NP → N 0.7
PP → P NP 1.0
N → people 0.5
N → fish 0.2
N → tanks 0.2
N → rods 0.1
V → people 0.1
V → fish 0.6
V → tanks 0.3
P → with 1.0

for i = 0; i < #(words); i++
    for A in nonterms
        if A -> words[i] in grammar
            score[i][i+1][A] = P(A -> words[i]);
```
// handle unaries
boolean added = true
while added
    added = false
    for A, B in nonterms
        if score[i][i+1][B] > 0 && A->B in grammar
            prob = P(A->B)*score[i][i+1][B]
            if(prob > score[i][i+1][A])
                score[i][i+1][A] = prob
                back[i][i+1][A] = B
                added = true
<p>| N → NP VP | 0.9 |
| S → VP | 0.1 |
| VP → V NP | 0.5 |
| VP → V | 0.1 |
| VP → V @VP_V | 0.3 |
| VP → V PP | 0.1 |
| @VP_V → NP PP | 1.0 |
| NP → NP NP | 0.1 |
| NP → NP PP | 0.2 |
| NP → N | 0.7 |
| PP → P NP | 1.0 |
| N → people | 0.5 |
| N → fish | 0.2 |
| N → tanks | 0.2 |
| V → people | 0.1 |
| V → fish | 0.6 |
| V → tanks | 0.3 |
| P → with | 1.0 |
| prob = score[begin][split][B] * score[split][end][C] * P(A → BC) |
| if (prob &gt; score[begin][end][A]) |
| score[begin][end][A] = prob |
| back[begin][end][A] = new Triple(split,B,C) |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fish</td>
<td>1</td>
<td>people</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>N → fish 0.2</td>
<td></td>
<td>NP → NP NP</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>N → N 0.14</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VP → V NP</td>
<td></td>
<td>VP → V NP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP → V 0.06</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>fish</td>
<td>1</td>
<td>people</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>N → fish 0.2</td>
<td></td>
<td>NP → NP NP</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>N → N 0.14</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VP → V NP</td>
<td></td>
<td>VP → V NP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP → V 0.06</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
</tr>
</tbody>
</table>

```python
//handle unaries
boolean added = true
while added:
    added = false
    for A, B in nonterms:
        prob = P(A->B)*score[begin][end][B];
        if prob > score[begin][end][A]:
            score[begin][end][A] = prob
            back[begin][end][A] = B
            added = true
```
<table>
<thead>
<tr>
<th>Rule</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td>0.9</td>
</tr>
<tr>
<td>S → VP</td>
<td>0.1</td>
</tr>
<tr>
<td>VP → V NP</td>
<td>0.5</td>
</tr>
<tr>
<td>VP → V</td>
<td>0.1</td>
</tr>
<tr>
<td>VP → V @VP_V</td>
<td>0.3</td>
</tr>
<tr>
<td>@VP_V → NP PP</td>
<td>1.0</td>
</tr>
<tr>
<td>NP → NP NP</td>
<td>0.1</td>
</tr>
<tr>
<td>NP → NP PP</td>
<td>0.2</td>
</tr>
<tr>
<td>NP → N</td>
<td>0.7</td>
</tr>
<tr>
<td>PP → P NP</td>
<td>1.0</td>
</tr>
<tr>
<td>N → people</td>
<td>0.5</td>
</tr>
<tr>
<td>N → fish</td>
<td>0.2</td>
</tr>
<tr>
<td>N → tanks</td>
<td>0.2</td>
</tr>
<tr>
<td>N → rods</td>
<td>0.1</td>
</tr>
<tr>
<td>V → people</td>
<td>0.1</td>
</tr>
<tr>
<td>V → fish</td>
<td>0.6</td>
</tr>
<tr>
<td>V → tanks</td>
<td>0.3</td>
</tr>
<tr>
<td>P → with</td>
<td>1.0</td>
</tr>
</tbody>
</table>

The diagram illustrates the parse tree for the sentence: "fish people fish tanks". The tree is constructed using a probabilistic context-free grammar (PCFG) with probabilities assigned to the rules. The tree is built by recursively applying rules to the non-terminal symbols (NP, VP, S) and their associated probability scores. The algorithm for building the parse tree involves iterating over possible splits in the sentence and selecting the split that maximizes the probability of the rule application. The score for each possible parse is calculated using the formula:

\[
\text{score}[\text{begin}][\text{split}][\text{end}][\text{C}] = \text{probability} \times \text{score}[\text{begin}][\text{split}][\text{end}][\text{B}] \times P(A \rightarrow BC)
\]

where A, B, and C are nonterminals, and the probability is the score of the rule application. The backpointer is stored in a new structure to trace back the parse tree. The algorithm continues until all possible splits are considered, and the tree with the highest probability is selected as the final parse.
for split = begin+1 to end-1
for A,B,C in nonterms
    prob = score[begin][split][B]*score[split][end][C]*P(A->BC)
    if prob > score[begin][end][A]
        score[begin][end][A] = prob
        back[begin][end][A] = new Triple(split,B,C)
<table>
<thead>
<tr>
<th></th>
<th>fish</th>
<th>1</th>
<th>people</th>
<th>2</th>
<th>fish</th>
<th>3</th>
<th>tanks</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>N → people 0.5</td>
<td>NP → NP NP</td>
<td>N → people 0.5</td>
<td>NP → NP NP</td>
<td>N → people 0.5</td>
<td>NP → NP NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V → people 0.1</td>
<td>VP → V NP</td>
<td>V → people 0.1</td>
<td>VP → V NP</td>
<td>V → people 0.1</td>
<td>VP → V NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP → N 0.35</td>
<td>VP → V 0.01</td>
<td>NP → N 0.35</td>
<td>VP → V 0.01</td>
<td>NP → N 0.35</td>
<td>VP → V 0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP → V 0.01</td>
<td>S → VP 0.001</td>
<td>VP → V 0.01</td>
<td>S → VP 0.001</td>
<td>VP → V 0.01</td>
<td>S → VP 0.001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S → VP 0.001</td>
<td></td>
<td>S → VP 0.001</td>
<td></td>
<td>S → VP 0.001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td>N → fish 0.2</td>
<td>NP → NP NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td>V → fish 0.6</td>
<td>VP → V NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td>NP → N 0.14</td>
<td>VP → V 0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td>VP → V 0.06</td>
<td>S → VP 0.006</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td>S → VP 0.006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>N → tanks 0.2</td>
<td>NP → NP NP</td>
<td>N → tanks 0.2</td>
<td>NP → NP NP</td>
<td>N → tanks 0.2</td>
<td>NP → NP NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V → tanks 0.1</td>
<td>VP → V NP</td>
<td>V → tanks 0.1</td>
<td>VP → V NP</td>
<td>V → tanks 0.1</td>
<td>VP → V NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP → N 0.14</td>
<td>VP → V 0.03</td>
<td>NP → N 0.14</td>
<td>VP → V 0.03</td>
<td>NP → N 0.14</td>
<td>VP → V 0.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP → V 0.03</td>
<td>S → VP 0.003</td>
<td>VP → V 0.03</td>
<td>S → VP 0.003</td>
<td>VP → V 0.03</td>
<td>S → VP 0.003</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for `split = begin+1` to `end-1`

for `A,B,C` in nonterms

```
prob = score[begin][split][B] * score[split][end][C] * P(A->BC)
```

if `prob > score[begin][end][A]`

```
    score[begin][end][A] = prob
    back[begin][end][A] = new Triple(split,B,C)
```

P → with 1.0
Call buildTree(score, back) to get the best parse
Constituency Parser Evaluation
Evaluating constituency parsing

Gold standard brackets:  S-(0:11), NP-(0:2), VP-(2:9), VP-(3:9), NP-(4:6), PP-(6:9), NP-(7,9), NP-(9:10)

Candidate brackets:  S-(0:11), NP-(0:2), VP-(2:10), VP-(3:10), NP-(4:6), PP-(6-10), NP-(7,10)
Evaluating constituency parsing

Gold standard brackets:
S-(0:11), NP-(0:2), VP-(2:9), VP-(3:9), NP-(4:6), PP-(6-9), NP-(7,9), NP-(9:10)

Candidate brackets:
S-(0:11), NP-(0:2), VP-(2:10), VP-(3:10), NP-(4:6), PP-(6-10), NP-(7,10)

Labeled Precision 3/7 = 42.9%
Labeled Recall 3/8 = 37.5%
LP/LR F1 40.0%
Tagging Accuracy 11/11 = 100.0%
How good are PCFGs?

- Penn WSJ parsing accuracy: about 73% LP/LR F1
- Robust
  - Usually admit everything, but with low probability
- Partial solution for grammar ambiguity
  - A PCFG gives some idea of the plausibility of a parse
  - But not so good because the independence assumptions are too strong
- Give a probabilistic language model
  - But in the simple case it performs worse than a trigram model
- The problem seems to be that PCFGs lack the lexicalization of a trigram model
Lexicalization of PCFGs

Introduction
(Head) Lexicalization of PCFGs

[Magerman 1995, Collins 1997; Charniak 1997]

- The head word of a phrase gives a good representation of the phrase’s structure and meaning
- Puts the properties of words back into a PCFG
(Head) Lexicalization of PCFGs

[Magerman 1995, Collins 1997; Charniak 1997]

- Word-to-word affinities are useful for certain ambiguities
  - PP attachment is now (partly) captured in a local PCFG rule.
    - Think about: What useful information isn’t captured?

- Also useful for: coordination scope, verb complement patterns
Lexicalized parsing was seen as the parsing breakthrough of the late 1990s

- Eugene Charniak, 2000 JHU workshop: “To do better, it is necessary to condition probabilities on the actual words of the sentence. This makes the probabilities much tighter:
  - $p(VP \rightarrow V \ NP \ NP) = 0.00151$
  - $p(VP \rightarrow V \ NP \ NP \mid \text{said}) = 0.00001$
  - $p(VP \rightarrow V \ NP \ NP \mid \text{gave}) = 0.01980$

- Michael Collins, 2003 COLT tutorial: “Lexicalized Probabilistic Context-Free Grammars ... perform vastly better than PCFGs (88% vs. 73% accuracy)”
Dependency Parsing

Introduction
Dependency syntax postulates that syntactic structure consists of lexical items linked by binary asymmetric relations ("arrows") called dependencies.

The arrows are commonly typed with the name of grammatical relations (subject, prepositional object, apposition, etc.).
Dependency syntax postulates that syntactic structure consists of lexical items linked by binary asymmetric relations ("arrows") called dependencies.

The arrow connects a head (governor, superior, regent) with a dependent (modifier, inferior, subordinate).

Usually, dependencies form a tree (connected, acyclic, single-head).
A dependency grammar has a notion of a head. Officially, CFGs don’t.

But modern linguistic theory and all modern statistical parsers (Charniak, Collins, Stanford, …) do, via hand-written phrasal “head rules”:

- The head of a Noun Phrase is a noun/number/adj/…
- The head of a Verb Phrase is a verb/modal/….

The head rules can be used to extract a dependency parse from a CFG parse.
Methods of Dependency Parsing

1. Dynamic programming (like in the CKY algorithm)
   You can do it similarly to lexicalized PCFG parsing: an $O(n^5)$ algorithm
   Eisner (1996) gives a clever algorithm that reduces the complexity to $O(n^3)$, by producing parse items with heads at the ends rather than in the middle

2. Graph algorithms
   You create a Maximum Spanning Tree for a sentence
   McDonald et al.’s (2005) MSTParser scores dependencies independently using a ML classifier (he uses MIRA, for online learning, but it could be MaxEnt)

3. Constraint Satisfaction
   Edges are eliminated that don’t satisfy hard constraints. Karlsson (1990), etc.

4. “Deterministic parsing”
   Greedy choice of attachments guided by machine learning classifiers
   MaltParser (Nivre et al. 2008) – discussed in the next segment
Dependency Conditioning Preferences

What are the sources of information for dependency parsing?

1. Bilexical affinities  
   [issues → the] is plausible

2. Dependency distance  
   mostly with nearby words

3. Intervening material
   Dependencies rarely span intervening verbs or punctuation

4. Valency of heads
   How many dependents on which side are usual for a head?

ROOT Discussion of the outstanding issues was completed.
MaltParser
[Nivre et al. 2008]

• A simple form of greedy discriminative dependency parser
• The parser does a sequence of bottom up actions
  • Roughly like “shift” or “reduce” in a shift-reduce parser, but the “reduce” actions are specialized to create dependencies with head on left or right

• The parser has:
  • a stack $\sigma$, written with top to the right
    • which starts with the ROOT symbol
  • a buffer $\beta$, written with top to the left
    • which starts with the input sentence
  • a set of dependency arcs $A$
    • which starts off empty
  • a set of actions
Actions ("arc-eager" dependency parser)

Start: $\sigma = [\text{ROOT}], \beta = w_1, \ldots, w_n, A = \emptyset$

1. Left-Arc $r$ 
   $\sigma|w_i, w_j|\beta, A \Rightarrow \sigma, w_j|\beta, A \cup \{r(w_j, w_i)\}$
   Precondition: $r'(w_k, w_i) \notin A, w_i \neq \text{ROOT}$

2. Right-Arc $r$
   $\sigma|w_i, w_j|\beta, A \Rightarrow \sigma|w_i|w_j, \beta, A \cup \{r(w_i, w_j)\}$

3. Reduce 
   $\sigma|w_i, \beta, A \Rightarrow \sigma, \beta, A$
   Precondition: $r'(w_k, w_i) \in A$

4. Shift
   $\sigma, w_i|\beta, A \Rightarrow \sigma|w_i, \beta, A$

Finish: $\beta = \emptyset$

This is the common "arc-eager" variant: a head can immediately take a right dependent, before its head is found.
Example

Happy children like to play with their friends.

1. Left-Arc\(_r\) \(\sigma|w_i, w_j|\beta, A \Rightarrow \sigma, w_j|\beta, A \cup \{r(w_i, w_j)\}\)
   Precondition: \((w_k, r', w_i) \notin A, w_i \neq \text{ROOT}\)
2. Right-Arc\(_r\) \(\sigma|w_i, w_j|\beta, A \Rightarrow \sigma|w_i|\beta, A \cup \{r(w_i, w_j)\}\)
3. Reduce \(\sigma|w_i, \beta, A \Rightarrow \sigma, \beta, A\)
   Precondition: \((w_k, r', w_i) \in A\)
4. Shift \(\sigma, w_i|\beta, A \Rightarrow \sigma|w_i, \beta, A\)
Example

Happy children like to play with their friends.

RA$_{xcomp}$ [ROOT, like, play] [with their, ...] $A_4 \cup \{xcomp(like, play) = A_5$

RA$_{prep}$ [ROOT, like, play, with] [their, friends, ...] $A_5 \cup \{prep(play, with) = A_6$

Shift [ROOT, like, play, with, their] [friends, .] $A_6$

LA$_{poss}$ [ROOT, like, play, with] [friends, .] $A_6 \cup \{poss(friends, their) = A_7$

RA$_{pobj}$ [ROOT, like, play, with, friends] [.] $A_7 \cup \{pobj(with, friends) = A_8$

Reduce [ROOT, like, play, with] [.] $A_8$

Reduce [ROOT, like, play] [.] $A_8$

Reduce [ROOT, like] [.] $A_8$

RA$_{punc}$ [ROOT, like, .] [ ] $A_8 \cup \{punc(like, .) = A_9$

You terminate as soon as the buffer is empty. Dependencies = $A_9$
MaltParser
[Nivre et al. 2008]

• We have left to explain how we choose the next action
• Each action is predicted by a discriminative classifier (often SVM, could be maxent classifier) over each legal move
  • Max of 4 untyped choices, max of $|R| \times 2 + 2$ when typed (label)
  • Features: top of stack word, POS; first in buffer word, POS; etc.
• There is NO search (in the simplest and usual form)
  • But you could do some kind of beam search if you wish
• The model’s accuracy is *slightly* below the best LPCFGs (evaluated on dependencies), but
• It provides close to state of the art parsing performance
• It provides **VERY** fast linear time parsing
Evaluation of Dependency Parsing:
(labeled) dependency accuracy

**Gold**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>She</td>
<td>nsubj</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>saw</td>
<td>root</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>the</td>
<td>det</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>video</td>
<td>nn</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>lecture</td>
<td>dobj</td>
<td></td>
</tr>
</tbody>
</table>

**Parsed**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>She</td>
<td>nsubj</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>saw</td>
<td>root</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>the</td>
<td>det</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>video</td>
<td>nsubj</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>lecture</td>
<td>ccomp</td>
<td></td>
</tr>
</tbody>
</table>

**Accuracy (Acc)**

\[
\text{Acc} = \frac{\# \text{ correct deps}}{\# \text{ of deps}}
\]

**Unlabeled/Labeled Attachment Score**

- **UAS**
  \[
  \frac{4}{5} = 80\%
  \]
- **LAS**
  \[
  \frac{2}{5} = 40\%
  \]
Representative performance numbers

- The CoNLL-X (2006) shared task provides evaluation numbers for various dependency parsing approaches over 13 languages
  - MALT: LAS scores from 65–92%, depending greatly on language/treebank
- Here we give a few UAS numbers for English to allow some comparison to constituency parsing

<table>
<thead>
<tr>
<th>Parser</th>
<th>UAS%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sagae and Lavie (2006) ensemble of dependency parsers</td>
<td>92.7</td>
</tr>
<tr>
<td>Charniak (2000) generative, constituency</td>
<td>92.2</td>
</tr>
<tr>
<td>Collins (1999) generative, constituency</td>
<td>91.7</td>
</tr>
<tr>
<td>McDonald and Pereira (2005) – MST graph-based dependency</td>
<td>91.5</td>
</tr>
<tr>
<td>Yamada and Matsumoto (2003) – transition-based dependency</td>
<td>90.4</td>
</tr>
</tbody>
</table>
Projectivity

- Dependencies from a CFG tree using heads, must be **projective**
  - There must not be any crossing dependency arcs when the words are laid out in their linear order, with all arcs above the words.
- But dependency theory normally does allow non-projective structures to account for displaced constituents
  - You can’t easily get the semantics of certain constructions right without these nonprojective dependencies

Who did Bill buy the coffee from yesterday?
Handling non-projectivity

• The arc-eager algorithm we presented only builds projective dependency trees

• Possible directions to head:
  1. Just declare defeat on nonprojective arcs
  2. Use a dependency formalism which only admits projective representations (a CFG doesn’t represent such structures…)
  3. Use a postprocessor to a projective dependency parsing algorithm to identify and resolve nonprojective links
  4. Add extra types of transitions that can model at least most non-projective structures
  5. Move to a parsing mechanism that does not use or require any constraints on projectivity (e.g., the graph-based MSTParser)
Dependency paths identify relations like protein interaction

[Erkan et al. EMNLP 07, Fundel et al. 2007]

\[
\text{KaiC} \xleftarrow{\text{nsubj}} \text{interacts} \xrightarrow{\text{prep\_with}} \text{SasA}
\]

\[
\text{KaiC} \xleftarrow{\text{nsubj}} \text{interacts} \xrightarrow{\text{prep\_with}} \text{SasA} \xrightarrow{\text{conj\_and}} \text{KaiA}
\]

\[
\text{KaiC} \xleftarrow{\text{nsubj}} \text{interacts} \xrightarrow{\text{prep\_with}} \text{SasA} \xrightarrow{\text{conj\_and}} \text{KaiB}
\]